Parallel Object-Oriented (amputation Applied to a Finite Element Problem

Jon B. Weissman and Andrew S. Grimshaw
Department of Computer Science
University of Virginia

Robert 1 'erraro
Jet Propulsion Laboratory/California Institute of Technology

Abstract

The conventional wisdom in the scientific computing community is that the best way to solve large-scale numerically -intensive scientific problems on today's parallel MIMD computers is to use Fortran or C programmed in a data-parallel style using low-level message-passing primitives. This approach inevitably leads to non-portable codes, extensive development time, and restricts parallel programming to the domain of the expert programmer. We believe that these problems are not inherent to parallel computing but are the result of the tools used. We will strew that comparable performance can be achieved with little effort if better 1001s that present higher level abstractions are used. The vehicle for our demonstration is a 2D electromagnetic finite element scattering code w c. have implemented in Mentat, an object-oriented parallel processing system. We briefly describe the application, Mentat, the implementation, and present performance results for both a Mentat and a ll;ind-coded parallel Fortran version.1

1.0 Introduction

Developing scientific applications on current parallel computers is difficult due to the absence of suitable programming tools and models to manage the complex details of parallel programming. The vast majority of today's systems are programmed in an arc]liicc(urc-specific way using low-level message-passing primitives that arc both hard to use and lead to non-portable codes. These systems are typically programmed in Fortran or C in a data-parallel SPMD style. W c believe that the problem is not the architectures, but the tools that have been used to program them, Wc will show in this paper that one can parallelize a real scientific application and obtain good performance with little effort if the right tools arc used.

The too] that we have used is Mentat[5], an object-oriented parallel processing system developed at the University of Virginia. Using Mentat, the user is responsible for identifying object boundaries and specify - ing those object classes that have sufficient computational complexity to warrant parallel execution. The Mentat compiler and run-time system arc responsible for managing al] aspects of communication, synchronization, and scheduling for the user. Mentat performs tasks that humans perform poorly, while the pro-

---

grammer performs tasks (data and program decomposition) that compilers perform poorly. Thus, Mentat exploits the capabilities of both compilers and humans. Mentat is currently available on a range of platforms from networks of heterogeneous workstations to tightly coupled machines such as the Intel iPSC/860. An important benefit of the Mental approach is that applications developed on one pl at form arc source code-por[ab]c from one plat form to another. This eliminates another problem common to writing soft ware for parallel architect ures, that applications arc not portable across pl at forms.

The vehicle for our demonstration is a 2D electromagnetic finite element scattering code (EM). The application was chosen for three reasons: 1) it is a real, non-t rivial, scient i fic code; 2) the sequential Fortran code was readily avail able; and 3) the application had previously been hand parallelized for a number of MIMD computers (Caltech/JPL Mark IIIfp Hypercube, Intel iPSC/860, and Intel Delta) using explicit message-passing primitives, providing us with the opportunity 10 compare the performance of hand generated parallelism against our compiled Mental version. The code computes the electric or magnetic field On an unstructured finite element mesh which defines the scattering objects as well as the space surrounding it.

Our work on the Mentat implementation of this code focuses on two issues: what is the overhead penalty which must be paid in order to use Mentat for this application, and how easy is it to apply Mentat to a scientific application like the finite element scattering code. What we have found is that the application domain mapped well to the object-oriented paradigm, and that the performance of the Mental version is comparable to the hand-coded version for small numbers of processors, but dots not scale as well in this initial implementation.

This paper is organized as follows. Section 2 discusses the EM application and finite element method. Section 3 provides an overview of Mentat. Section 4 discusses the objcc[-oriented redesign of the EM application. Section 5 describes the parallel EM implementation via Mentat. Section 6 presents some preliminary results obtained with the Mental version, and Section 7 provides a summary and future work.

2.0 The EM Problem

The finite element method (FEM) has been in use for many years in structural mechanics [10] and has become popular in recent years as a technique for use on electromagnetic problems [13]. FEM has the advantage of being able to deal with the specific geometry of objects by using unstructured gridding which follows an object's shape. This can be of particular importance in EM scattering problems, where the correct representation of a scatterer's surface is necessary for accurate computation. Finite elements arc used in 2 and 3 dimensional electromagnet ic scattering problems to model objects of complex composition. The "hand-coded" FEM code has been implemented On several MIMD computers, using explicit message passing. A complete description of this co(Jc, along wilh parallel implementation description and performance, is found in [3]. For this work, we have concentrated on a 2D FEM problem.

The general scattering problem solved by the 2D EM code is illustrated in Fig. 1. The code solves a
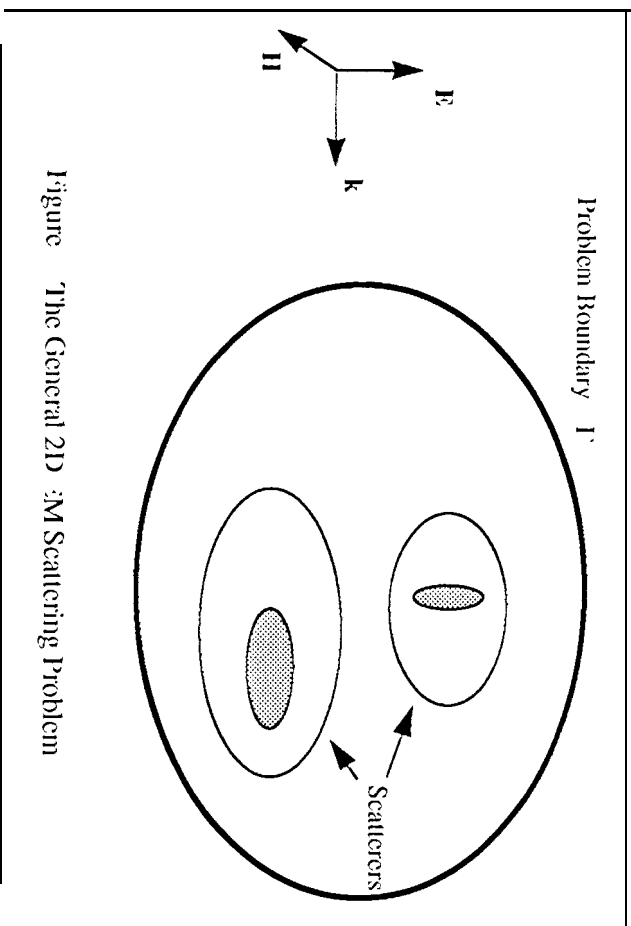
Problem Boundary Γ

Figure 1   The General 2D EM Scattering Problem

Helmholtz equation for the electric or magnetic fields in the vicinity of a set of scatterers:

$$\nabla \bullet (-\frac{\nabla E}{\mu}) + \frac{\omega^2}{c^2} \varepsilon E = 0 \qquad (1)$$

Here E is the electric field, and μ and ε are materials constants. An absorbing boundary condition on the boundary Γ uniquely specifies the problem. The finite element method solves the equivalent "weak form" integral equation:

$$\int_{\Omega} d^2v \left[-\frac{\nabla T \bullet \nabla E}{\mu} + \frac{\omega^2}{c^2} \varepsilon ET \right] = \int_{\Gamma} ds \frac{T}{\mu} \frac{\partial E}{\partial n} \qquad (2)$$

The boundary condition is incorporated into the surface integral on Γ.

The 2D integral equation is transformed into a set of linear equations by decomposing the problem domain into a set of finite elements. The problem domain Ω is meshed with nodal points at which the solution is to be found, matching the geometry of the objects. These nodes are then tiled with a set of finite elements as in Fig. 2. In 2D, the elements might be triangles or quadrilaterals. A set of basis functions are defined at each node in the mesh, which have nonzero value only within the elements of which it is a part. These basis functions are generally some polynomial function which is 1 at the node defining it, 0 at all other nodes in the element, and 0 along the edges of the element opposite the defining node. An example of a linear basis function at a node in a section of finite element mesh is given in Fig. 3. The function is continuous inside and across elements, dropping to zero at the element edges which do not intersect the node. On all other elements in the grid, the basis function is identically zero.
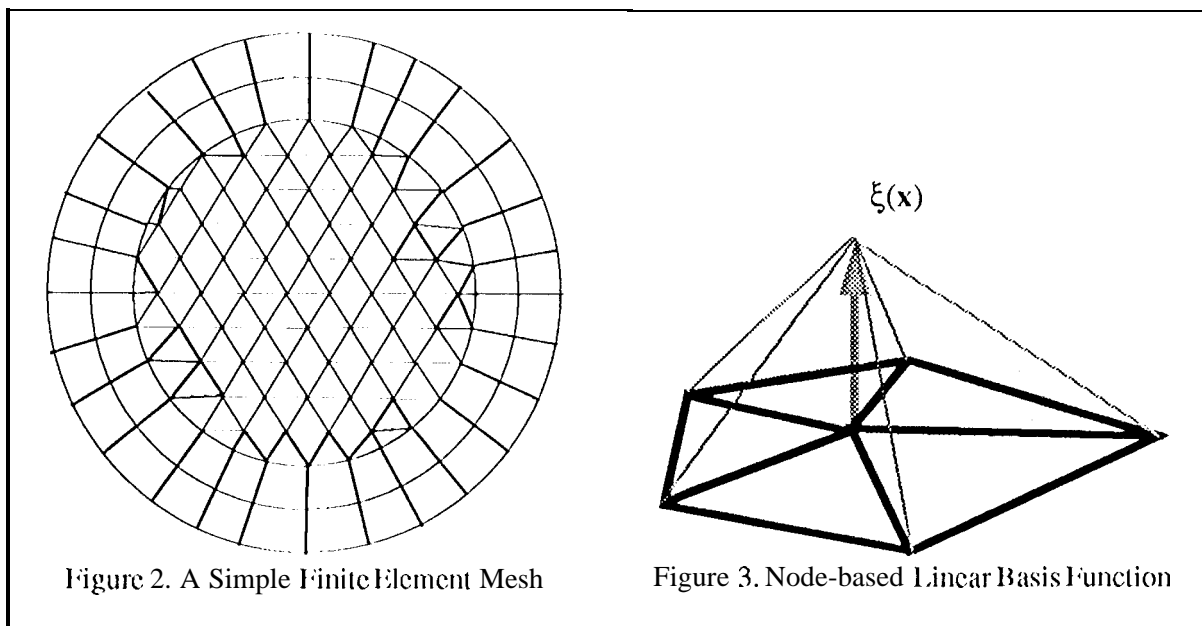
3

Figure 2. A Simple Finite Element Mesh   Figure 3. Node-based Linear Basis Function

The field quantities (electric or magnetic) may be expressed as a linear combination of these basis functions:

$$E(x) = \sum_i d_i \xi_i(x) \tag{3}$$

where, $\xi_i(x)$ is the basis function at the ith node, and d_i is its coefficient in the representation for E. Notice that since, by definition, all other basis functions are () at node i, the value of d_i is in fact the value of E at the ith node. We also write the test function T in terms of these basis functions. Substituting these into Eq. (2), and recognizing that the test function T must be arbitrary results in a matrix equation for the field coefficients d_i:

$$\bar{K} \cdot d = F \tag{4}$$

where d is the vector of field coefficients d_i from Eq. (3), and K and F, known as the stiffness matrix and force vector, are given by expressions involving integrals of individual basis functions or products of basis functions. Since all basis functions are localized to a handful of finite elements, these integrals are non-zero only for those elements which contain the basis functions involved. This results in a K matrix which is quilt sparse. As a matter of practice, these integrals are computed on an element by element basis, with each element's contribution to $\overline{K}$ and F added in its turn. In this manner, the complexity of integrating over a domain of irregular geometries is reduced to integrating over a set of regular finite sized elements.

This is the basic finite element method. The EM finite element application consists of two primary computation phases: 1) matrix assembly and 2) matrix solve. In matrix assembly, the finite elements compute contributions (i.e. matrix values) that are assembled (i.e. added) into the stiffness matrix $\overline{K}$. The stiffness matrix is banded and symmetric, in addition to being very sparse. During matrix assembly, the force VCC1OP'

4

F is also computed by the elements. This vector becomes the right-hand-side vector during the matrix solve compilation.

in matrix solve, the s ystem of equat ions represented by the stiffness matrix with the force vector as the. rigl](.hand-side, is solved by a conjugate-gradient algorithm known as Ili-conjugate gradient[11]. The algorithm requires three basic operations: matrix-vector multiplication, vector dot product, and vector saxpy. 'l he solve phase poses challenges to achieving good perform ance on parallel machines due to the sparse nature of the malrix-vector operations.

3.0 Mental overview

Mental is a parallel object-orien(cd programming environment developed at the University of Virginia. Mental was designed to address two problems that plague programming parallel MIMD architectures. First, writing parallel programs by band is very difficult. The programmer must manage communication, synchronization, and scheduling. The burden of correctly managing the environment often overwhelms programmers, and requires a considerable investment of time and energy. Second, once implemented on a particular MIMD architecture, the resulting codes arc usually not portable. Thus, considerable effort must be re-invested to port the application to a new architecture.

Mental offers a solution to these problems by providing: 1) easy-to-use paralleli sm; 2) high performance via parallel execution; and 3) applications portability across a wide range of platforms. The premise under-lying Mental is that writing programs for parallel machines dots not have to be hard. Instead, it is the lack of appropriate abstractions that has kept parallel architectures di fficult to program, and hence, inaccessible to m ainstream, production system programmers.

The Mental approach exploits the objccl-oriented paradigm to provide high-level abstractions that mask the complex aspects of parallel programming, communication, synchroni zation, and scheduling from the programmer. Instead of worrying about and managing these details, the programmer is free to concentrate on t t ie details of the application. The programmer uscs application domain knowledge to speci fy those object classes (Mental classes) that arc of sufficient computational complexity to warrant parallel execu-tion. The remaining complex tasks arc handled by Mental.

'1'here arc two primary components of Mental: the Mental Programming Language (Ml'],) [6] and the Mental run-time system [7]. MPL is an object-orien(cd programming language based on C++[14] that masks the complexity of the parallel environment from the programmer. Ment at cl asses consist of con-tained objects (local and member variables), their procedures, and a thread of control. Instances of Mental classes, known as Mental objects, arc the computation grains, Because Mental is based on a layered virtual machine model, and each layer introduces some amount of overhead, Mental classes must be medium-to-large grained to mask these overheads.

Mental classes are denoted by the inclusion of the keyword "mental" in the class definition, as in the mental class sparse_ worker shown below. Mental classes may be defined as either persistent or regular,

```
persistent mental class sparse worker (
///  private data and function members
public:
     complexvec* m vec mul t (complexvec* vec);
     ...
);
```

Instances of regular Mental classes arc logically stateless, thus tbc implementation may create a new instance to handle every member function invocation. Persistent Mental classes maintain state information between member function invocations. This is an advantage for operations that require large amounts of data, or that require persistent semantics. Instances of Mental classes are used exactly like C++ classes, as in the fragment below. One difference is that persistent Mental objects are instantiated by the *create* com -

```
sparse worker worker;
...
worker.create ();
result = worker.m vec mul (rhs vec);
```

mand.

Mental supports a notion of parallelism encapsulation. Parallelism encapsulation takes two forms that we call intra-object encapsulation and inter-object encapsulation. Intra-object encapsulation of parallelism means that callers of a Mental object member function arc unaware of whether the implementation of a member function is sequential or parallel. Inter-object encapsulation of parallelism means that programmers of code fragments (e.g., a Mental object member function) need not concern themselves with the parallel execution opportunities between the different Mental object member functions they invoke. Thus, the data and control dependencies between Mental class instances involved in invocation, communication, and synchronization arc automatically detected and managed by the compiler and run-time system without further programmer intervention.

The computation model underlying Mental is the macro data flow model [7], a large-grain, graph-based, data-driven computation model. The Mental run-time system supports the macro data flow model via the provision of a virtual macro data flow machine. Because the compiler uses a virtual machine model, porting applications 10 a new architecture does not require any user source level changes. Once the virtual machine has been ported, user applications arc re-compiled and can execute immediately.

Mental rum on Sun 3s, Sun 4s, the Intel iPSC/2 and iPSC/860, and the Silicon Graphics Iris. We arc currently porting Mental to the TMC CM-5 and the Intel Delta (Paragon). Performance results on a range of applications arc available, and are quite encouraging [8].

4.0 Object-Oriented EM Design

Converting the sequential Fortran EM code into a parallel object-oriented code via Mental requires two major steps: 1) porting the existing code to a sequential object-oriented language (C++) and 2) porting the

C++ implementation to Mental. Both 1) and 2) pose different challenges. The Mental conversion will be discussed in the next section. The C++ conversion requires a redesign or "paradigm shift" from the Fortran domain to the object-oriented domain. This is a non-trivial conversion even before opportunities for parallelism are considered. The use of global structures, aliasing, and the lack of data abstraction in Fortran made the transition to C++ challenging.

4.1 The Approach

There are several approaches that can be taken to convert an existing Fortran implementation to C++, and of these we considered two candidates: 1) reuse the Fortran artifact and wrap C++ classes around the existing code and 2) redo everything in C++. Although we operated on a very short time budget, we felt that option 2) was clearly the best choice since this affords us the greatest flexibility in experimenting with different problem decompositions. We also believe that a "pure" object-oriented system is easier to extend than a mixed-language implementation. Since this work is part of a continuing research effort, this is an important factor. The down-side is a performance loss experienced in the sequential object-oriented code due to disparities in current compiler technology and the quality of existing Fortran numeric libraries. We expect this gap to close in the future, however.

Our approach then was to treat the Fortran code as a functional specification of the behavior of the FM application at an algorithmic level. For example, the element computation implementation in C++ mirrored the Fortran. The Fortran code was used as a guide or reference for low-level details and the mathematics. At the high-level, we selected a class-object hierarchy that reflected our knowledge of the problem domain. Once a natural class structure was established, we inspected the Fortran code for algorithmic details that were necessary to faithfully reimplement the numeric computations encapsulated in the C++ member functions.

During this conversion the main difficulty was the lack of data abstraction in the Fortran code - this is not surprising since Fortran doesn't support this notion. The data structures were typically represented as collections of separate arrays. Data that was logically connected had to be inferred by its use or by the code comments. An example of this was the absence of an explicit representation for the elements themselves. The implicit element representation was scattered across numerous global arrays. The object-oriented approach requires just the opposite: that logically-connected data be represented together and encapsulated. Another difficulty with the conversion was the use of complex numbers extensively in the Fortran code. We had to implement a fairly extensive complex class in C++. Our implementation was less efficient than the built-in, optimized, complex data type provided by Fortran.

One of the research objectives of this work is to consider the effort involved in converting existing scientific applications to an object-oriented platform (C++) and then to Mental. The port from the sequential Fortran implementation to a fully tested C++ code took two graduate students six weeks (about one man-month of effort). Part of this time was needed to gain familiarity with the problem domain, which was unfamiliar, and to review the details of Fortran. We feel that the short time frame validates our decision to implement a C++ version and also provides further evidence for the suitability of the object-oriented para-

Figure **4.** Finite element class hierarchy
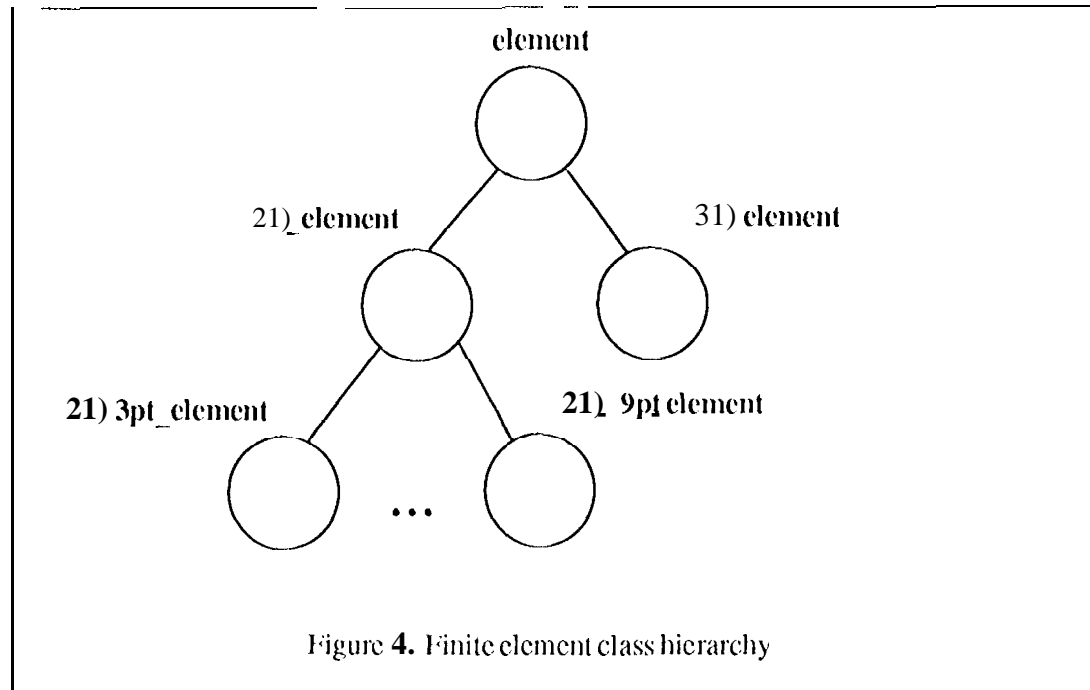
digm as applied to scientific applications like the EM problem. However, while this evidence suggests a good fit between the object-oriented paradigm and this particular problem domain, there are performance trade-offs. These are discussed in the final section.

4.2 C++ Classes

The heart of the sequential EM implementation is its decomposition into C++ classes. Some of the C++ classes will become parallel or Mentat classes in the parallel EM implementation. Discussion of Mentat classes is deferred until the next section. The problem domain can be broken down into two phases: element assembly and matrix solve. The class-object hierarchies reflect this decomposition.

The first phase involves the finite element computations needed to construct the sparse stiffness matrix and right-hand-side vector. During this phase, each element computes a contribution to the matrix. We represented the elements as C++ objects contained within a finite element class hierarchy, shown in Fig. 4.

The hierarchy is rooted by the virtual base class *element* and the derived classes reflect the different types of finite elements that are used in EM problems. The element type depends both on the physical characteristics of the material (e.g. 2D/3D or triangle/quadrilateral), and on the way the element computes its matrix contribution (e.g. 3pt/6pt quadrature). A part of the C++ specification for the finite element hierarchy is shown in Fig. 5.

The element representation is simply the nodal points that define its boundaries. The derived classes contain element-specific information, such as the basis functions, that are needed in the element computations. The element contributions are computed by *get kf* and are assembled into the sparse stiffness matrix during the first phase. This sparse matrix is stored as a list of sparse vectors, each row is represented by the

8

```
/ /  Base class of the element hierarchy
class element {
    int *nodes; // nodal points
    int num nodes;
    ...
public:
    // returns matrix and force-vector contributions
    virtual KF_contrib* get_kf ();
    element ();
};
```

```
/ / 3pt triangle 2D element
class 2D 3pt element  : 2D element {
// basis" frls
...
public:
    KF_contrib* get kf ()  ;
    2D 3pt element '(int *nodes,...) ;
    ...
};
```

Figure 5. Finite element class specification

sparse vec class. The sparse matrix is a special class known as a Mentat class and this is discussed in the next section.

During the second phase of the computation, matrix solve, "dense" vectors of complex **numbers** are computed by the application. The representation of complexvec is a memory-contiguous variable-sized array of complex type. The *complexvec* class specification is given in Fig. 6. Memory contiguity is important in the parallel domain for objects that are transported between address spaces (such as objects of type complexvec),

Both the sparse. vec and complexvec classes had been implemented previously and we were able to reuse them with slight modification (to use complex numbers), The C++ classes form the basis for the parallel EM design. The remaining C+ -t classes in the application have a dual role: these classes can be treated as C-t+ classes as in lilt sequential version or as Mentat classes in the parallel version. These arc discussed next. In the next section we will also show how everything fits together in the parallel EM implementation.

S.() Mentat EM Design

The design decisions that guided the transformation from the sequential Fortran implementation to the sequential C++ were motivated by three factors: 1) flexibility/extensibility, 2) fidelity and 3) support for

9

```
class complexvec : public DD array {
    int start_index, range;
    // memory continguous representation
    complex a[];
public:
    complexvec (int cols) ;
    complexvec* saxpy(complexvec *f ,  complex& m) ;
    complex vec* ssxpy(complexvec *f , complex& m) ;
    complexvec* dot product (complexvec * f) ;
    . . .
};
```

Fig. 6. Complexvec class specification

parallelization. Points 1) and 2) have been discussed and this section addresses point 3), transitioning to the parallel EM code.

The parallel EM code is based on the paral IcI objcxl-orient cd model of computat ion provided by Mental. While the design of the parallel EM code is concerned with points 1) and 2) above, it is driven by performance and scalability. in the parallel domain, the most critical factors affecting performance are computation granularity and load balance. In Mental, computation granularity is specified via a mechanism known as Mental classes, and load balance is achieved by an even partitioning of work across the instantiated Mental objects.

The parallel EM system design based on Mental is illustrated in Fig. 7. The assembl y (phase 1) and solve computations (phase 2) arc shown. The remainder of this section will describe the Mental classes usc(J, the rationale for choosing them, and other important details of the parallel EM design and Mental implement alien.

The selection of Mental classes is based upon exploiting opportunities for parallelism and achieving an acceptable computation granularity given Mental overheads and the characteristics of tile target architecture. Our target is the Intel iPSC/860, a very unbalanced machine in which communication costs dominate computation costs by several orders of magnitude. The Mental classes will need to be "computationally heavy", i.e. large-grainc(i, to achieve reasonable, performance give.n these factors. The EM application performs two main computations, element assembly and matrix solve, and time will be implemented via Mental classes.

J 'or element assembl y, there arc many opportunit ics for parallelism since the element computat ions arc independent and may proceed in parallel. To exploit maximal parallelism, we would turn tile finite element classes (of Fig. 4) into Mental classes. J lowever, a single element assembly computation is too fine-grained

Figure **7.** Parallel EM architecture
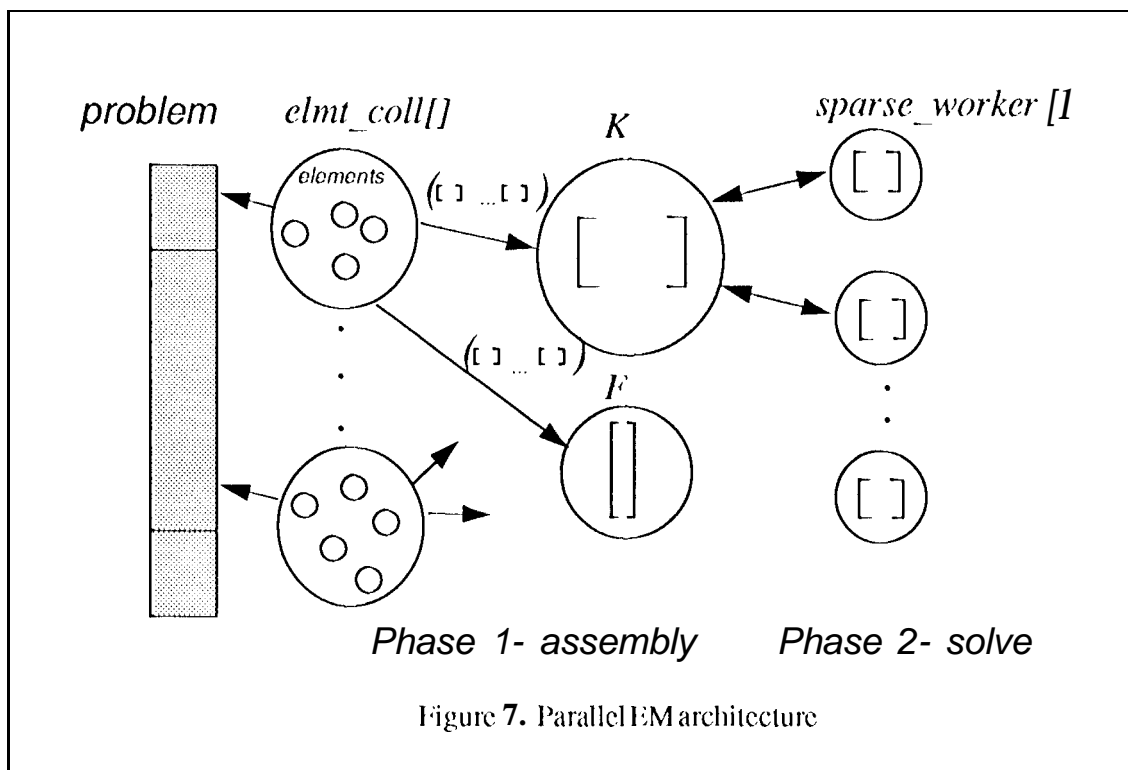
```
persistent mental class elmt_coll {
     element** elements;
     int element_num;
public:
     // element setup and partitioning
     void initialize (string *f, int i, int num_coll);

     // compute and assemble all element s
     void assemble (sparse matrix*K, svector*F);
     ...
};
```

Figure 8. Mental class elmt_coll specification

for Mental and this will lead to unacceptably poor performance. Instead, we define a Mental class which computes the contributions for a collection of elements, *elmt coll*, see Fig. 8. Notice that the e 1 mt coll class contains C+ + objects (of t ype element) as part of its representation.

A number of e 1 mt co 1 objects are instantiated at runtime and each computes in parallel. Each elrnt coll is assigned enough elements to achieve an acceptable computation granularity, The number

11

```
persist.ent mentat class sparse matrix {
    // sparse_matrix representation
    sparse vec** mat.ix;
    int size;
    ...
    // sparse_ worker information
    int num workers;
    sparse worker*workers;
    ...
public:
    // Each elmt_coll assembles to matrix
    void assemble (K list* K contrib);

    // Solve matrix equation using rhs vector F
    void solve (svector*F);

    // Set up matrix with number of workers
    void initialize (int num workers);
    ...
};
```

Figure 9. Mentat class sparse_ matrix specification

of elmt coll objects instant iated and how the individual elements get assigned to a particular elmt coll are discussed later. Once the elmt colls compute the matrix contributions and right-hand-side force-vector values associated with their contained elements (via *get_kf*), these values must be assembled into the stiffness matrix and force-vector respectively. The *assemble* member function defined on elmt coll initiates the element computations and invokes an assemble operation on the matrix. The Mentat class *sp arse matrix* represents the stiffness matrix, see Fig. 9. Matrix assembly is performed via the member function *assemble* called by each elmt coll. The definition of the Mentat class *svector*, the force-vector, is omitted.

Most of the computation time is spent in the matrix solve phase. The solve computation is performed by an iterative preconditioned Bi-conjugate gradient algorithm (BCG) [11] implemented by the *solve* member function of the Mentat class sparse matrix. Our implementation exploits the most profitable opportunity for parallelism in the algorithm, namely the sparse matrix-vector multiplications done in each iteration of the BCG algorithm.

Parallelizing the matrix-vector multiplications requires that another class Mentat class, *sparse worker*. be defined, see Fig. 10. The sparse worker class is responsible for performing matrix-vector multipl-cation on disjoint regions of the sparse matrix. A number of sparse worker objects are instantiated at runtime, and the sparse matrix is partitioned into row-contiguous regions and distributed to the

```
persistent mental class sparse_worker {
    // sparse worker representation
    sparse vec** my rows;

    // region of global matrix stored by worker
    region my reg;

    // partial result for matrix-vector multiply
    complexvec* result ;
    ...
public:
    // Distributes rows to worker
    void initialize (sparse_vec list* sparse rows, ... ) ;

    // Sparse mvec multiplication
    complexvec* m vec mult (complexvec* vec) ;
    ...
};
```
Figure 10. Mental class sparse_worker specification

sparse worker objects. This is done via the sparse_worker member function *initialize*. Once the sparse matrix has been distributed fully to the sparse worker objects, the sparse matrix object engages the sparse workers in parallel malt'ix-vector multiply operations (via m *vec_mult*) repeatedly during the solve phase. The sparse workers are encapsulated within the sparse matrix (Fig. 9) and this has performance implications as we will scc.

The Mental classes, sparse matrix, elmt col 1, and sparse worker, reflect the computationally-intensive phases of the application and result in a granularity suitable both for Mental and the target architecture. These classes also allow sufficient parallelism in the application to be exploited. One important advantage of Mental is that the serial EM code requires only a few "ifdefs" to turn these Mental classes into C++ classes (scc Fig. 11) - under 20 lines of code are unique to either the serial or parallel version.

At runtime, the programmer specifics the number of elmt col1 objects for the assembly phase and the number of spar se wor ker objects for the solve phase. The number of objects should match the total number of available processors assuming the granularity is sufficient. For large applications, this is usually the case. On the iPSC/860 under NX, only one object (i.e. process) may be placed on a processor Since these phases are non-overlapping (i.e. solve dots not begin until assembly has completed), the number of elmt col 1s and sparse workers will be the same.

Achieving acceptable performance in the parallel EM code depends upon good load balance. It is sufficient to load balance the assembly and solve phases separately since they are independent - only ii synchniz.alien between these phases is needed. The assembly load balance requires that the e 1 mt_col1 objects

```
#ifdef Mentat
persistent mentat class sparse_ worker {
#else!
class  sparse. worker {
#endif
// as above
...
};
```

Figure 11. Dual Mentat/C++ class specification using "ifdefs"

each perform about the same amount of computation. An even partition of the elements across the elmtcol1s would seem to be an easy solution. However, the general EM problem will contain elements of different types - more complex elements require more computation to determine matrix contributions. A good load balance solution ensures that each elmt_ coil has roughly the same number of elements of each type. As as approximation to this, our implementation randomizes the element input files and randomly assigns elements to the elm_ coil objects.

Similar] y, load balancing the solve phase requires that the sparse worker objects are evenly balanced for the matrix-vector multiplications. An even partition of the sparse mat r i x across the sparse workers may not lead to load balance since the matrix has non-uniform sparsity (i.e. the number of non-zeros per row differs) and only non-zero positions of the matrix will be multiplied by the sparse workers. Instead, load balancing is achieved by ensuring that each sparse worker has about the same number of non-zeros in the matrix region that it has been assigned. Note that the number of rows assigned to each sparse worker will, in general, be different.

The initial parallel EM design has a number of flaws that limit the scalability and performance of the system. The most obvious as seen clearly in Fig.7 is that a single sparse matrix object is a bottleneck for both matrix assembly and matrix solve. During the solve phase, partial results from the matrix-vector multiplies are fanned into the sparse matrix, thus creating a communication bottleneck. The problem is due to the encapsulation of the sparse workers within the sparse matrix. This is a classic problem with the object-oriented paradigm. The single sparse matrix object also limits the size problems that can be run since the entire matrix is assembled in one address space before it is distributed to the sparse workers. Furthermore, no attempt was made to parallelize the dot products that occur within the BCG loop. These dot products are a good source of parallelism, especially for large vectors. Nonetheless, the perform ance results are good when compared to a hand-coded version that does not suffer from these limitations. The results are presented in the next section.

Once the C++ version of the EM code had been fully implemented and tested, the Mental version (about 3000 lines of code) took about two weeks to complete. One of the major problems we had with Mentat on the iPSC/860 was the need to force arithmetic operands to be double-word aligned to get good perfor-

14

Figure 12. Parallel EM performance with Mental
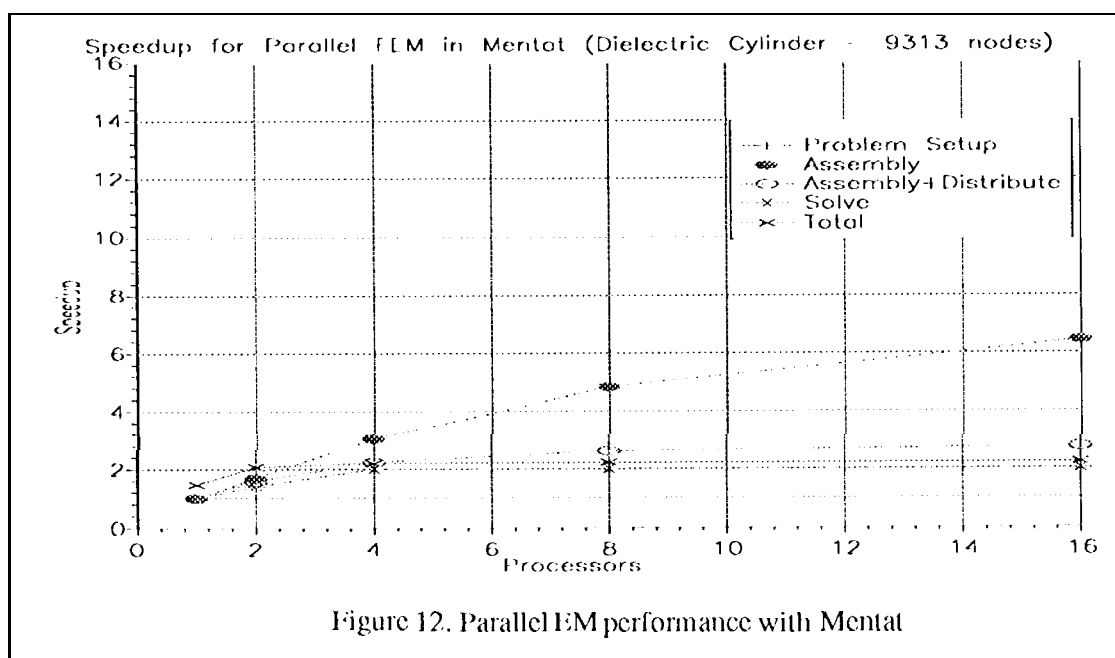
mance on Ibis machine. This required some low-level pointer code and was time-consuming to implement and test. The memory bottleneck imposed by the single spa r se mat rix object did not allow our EM problem to fit on a 8MB iPSC/860 node at ORNL, a 128 node machine. We eventually ran on a 16MB/node iPSC/860 at Caltech. Fortunately, the Mentat system binaries ported smoothly to the Caltech machine - no recompilation of the Mentat system code was necessary.

6.0 Preliminary Results

The initial Mentat EM code was developed on an tl-node Intel iPSC/860 at JPL and run on a 64-node Intel iPSC/860 at Caltech. The data collected are from an EM application that consisted of 2304 9pt quad-rilateral elements (9313 nodes). This is considered a small problem. We computed speedups with respect to the sequential C+ + EM code run on a single i860 node, see Fig. 12.

The results are divided into the several phases: 1) problem setup is the time taken for thee] mt _ co] 1s to read the element files from CFS and create the element partitions, 2) assembly is the time taken for the e 1 mt co] 1s to complete the matrix assembly operations, 3) assembly and distribute includes the time to distribute the matrix out to the workers, 4) solve is the time taken for the matrix solve operation, and 5) total is the total time taken by the application. We should reiterate that virtually no optimization of the Mentat version had been performed.

Our results are compared with a han(i-coded optimized parallel Fortran EM implementation that has been in development for some time. We expected the performance to be worse than the ban(i-coded version, but how much? The results indicate that this is indeed the case, but speedups were achieved even though the problem was small and the given implementation limitations that have been discussed, see Fig. 13.

15

Figure 13. Comparison of Mentat to Hand-coded EM version
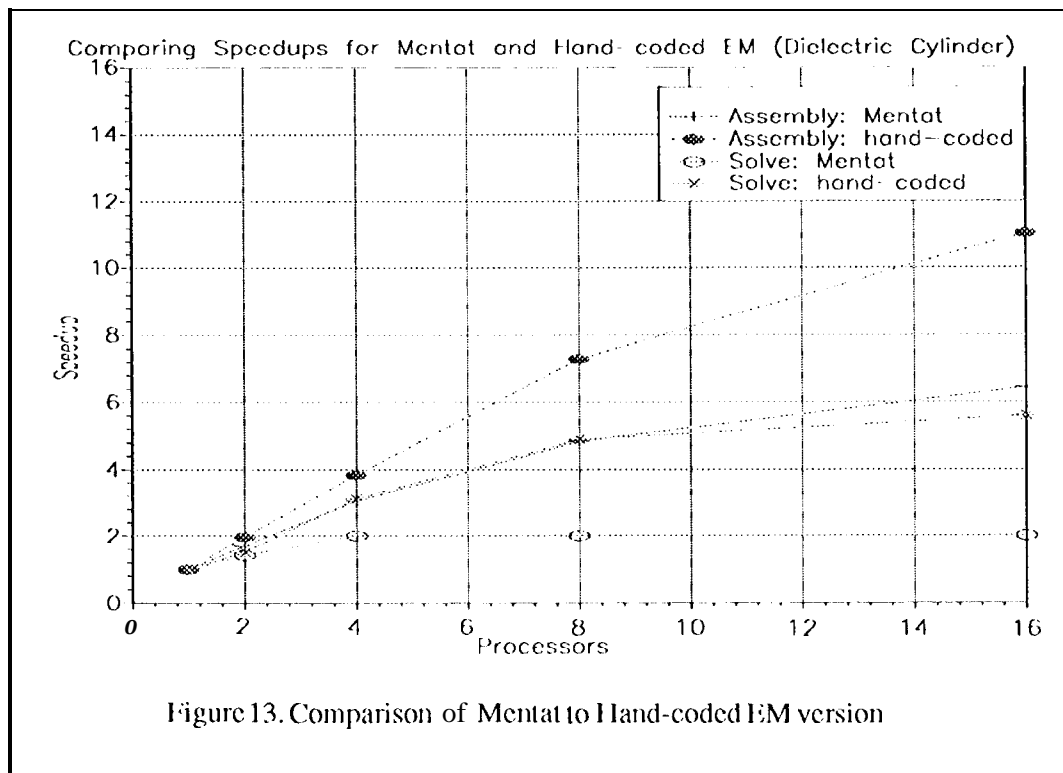
Comparison of the Mentat and the hand-coded versions indicates that the Mentat implementation is competitive with the band-coded for small numbers of processors, but that performance doesn't scale well as the number of processors is increased. This is due to the limitations that have been discussed, namely the **sparse** matrix bottleneck for assembly and matrix-vector communication, and the sequential dot products in the solver. It is not surprising that the initial Mentat version does not scale given the design.

7.() Summary and Future Work

The early results of the research are encouraging. The initial design and implementation of the parallel EM code using Mentat took under two months, including the time to perform the "paradigm shift" from Fortran to C++. This indicates to us that the parallel object-oriented model in general, and Mentat in particular, is well-suited to this problem domain. We have validated only part of this claim: that the EM problem has a natural representation in a object-oriented framework and we have provided further evidence that Mentat is an easy-to-use programming environment for developing parallel object-oriented scientific applications. However, the other part of the claim is that performance does not suffer greatly with Mentat when compared to the hand-coded version. Until the limitations with the current Mentat design are removed, this claim cannot yet be made.

Other researchers have begun to report on the experience of using object-oriented implementation techniques for scientific problems [1,4]. Our experience is similar to [1] in that programmer efficiency seems to be a more clear benefit than execution efficiency at present, The authors [1] report that the C++ perfor-

mance is within an order of magnitude of the Fortran code (ours is within a factor of 2-3), and that this provides some hope.

While the performance was not as good as the hatld-coded version, we have identified several performance enhancements that arc being incorporated into the next parallel EM design. The most important of these *is the* elimination of the sparse matrix bottleneck - the matrix will be assembled in a distributed fashion by the sparse workers directly. This will speed up assembly and eliminate the need to explicitly distribute the matrix out to the workers. We arc also planning to distribute the dot products, parallelizing the inner products that occur in the BCG loop, thus eliminating the fan-in communication bottleneck to the **sparse mat. r i x**. Instead, the spar se workers will interact in a more tightly-coupled fashion. Some encapsulation will be traded for performance. These changes will allow us to run much larger problems, and should yield a noticeable improvement in performance and scalability.

Acknowledgment s

8.0 References

[1] I. G. Angus, and J. 1.. Stolzy, "Experiences in Converting an Application from Fortran to C++: Beyond f2c," *C+-t at Work* Conference, November 1991.

[2] R. D. Ferraro, A. S. Grimshaw, and J. B. Weissman, "Object-oriented Parallelism Applied to a Finite-Element Code", poster presented at *Supercomputing* **'92,** Minneapolis Mn.

[3] R. D. Ferraro, "Solving PDEs for Electromagnetic Scattering Problems on Coarse Grained Concurrent Computers", in the PIER volume CO MPUTATIONAL ELECTROMAGNETICS AND SUPERCOMPUTER ARCHITECTURE, Elsevier Science Publishing Company, Inc. (Chapter, In Press).

[4] D. W. Forslund et al, "Experiences in Writing a Distributed Particle Simulation Code in C++ ," *Usenix C+ + Con* ference, 1990.

[5] A. S. Grimshaw, "Easy to Use Object-Oriented Parallel Programming with Mentat," to appear in *IEEE Computer*, May, 1993.

[6] A. S. Grimshaw, E. Loyot Jr., and J. B. Weissman, "Mentat Programming Language (MPL) Reference Manual," University of Virginia, Computer Science TR 91-32, 1991.

[7] A. S. Grimshaw, "The Mentat Run-Time System: Support for Medium Grain Parallel (amputation," *Proceedings of the Fifth Distributed Memory Computing Conference*, Charleston, SC., 1990.

[8] A. S. Grimshaw, W. '1'. Strayer, and P. Narayan, "The Good News About Dynamic Object-oriented Parallel Processing," University of Virginia, Computer Science TR 92-41, 1992.

[9] A. S. Grimshaw, E. A. West, and W. Pearson, "NO Pain and Gain! - Experiences with Mentat on Biological Application," *Proceedings of the First Symposium on High-Perfo rmance Distributed Computing*, Syracuse, NY, Sept., 1992.

[1()] '1'. J. R. 1 lughes, *The Finite Element Method*, Prentice-Hall, inc., Englewood Cliffs, New Jersey 1987.

[11] D. A. 11. Jacobs, "The Exploitation of Sparsity of Iterative Methods," *Sparse Matrices and 7'heir Uses*, edited by 1. S. Duff, Academic Press, London, 1982.

| 12] J. W. Parker, '1'. Cwik, R. D. Ferraro, P. C. Liewer, P. Lyster, and J. E. Patterson, "Helmholtz Finite 1 lements Performance On Mark 111 and InteliPSC/860 Hypercubes," *Proceedings of the Sixth Distributed Memory Computing Conference*, IEEE Computer Societ y Press, 1991.

[13] R. 1'. Silvestri and R. L. Ferrari, "Finite Elements for Electrical Engineers," Cambridge University Press, New York, 1983.

| 14111. Stroustrup, *The (,'++ Programming Language*, 2nd cd. Addism-Wesley, Reading, Mass., 1991.